Neural Network Do-it-Yourself

A Python library implementing different architectures of neural network for Machine Learning tasks

Damien Dam
Enzo Durand

April 16, 2022

Introduction

Neural network has been playing an important role in the field of Machine Learning as it is able to solve many problems ranging from classification to continuous prediction and image recognition. More complex networks even have super-human performance in various tasks such as stock trading or game playing.

With the help of the Python programming language and some third-party libraries, an average user today can easily build and deploy a neural network that runs complex learning algorithms within minutes. Our work in this project also aims to provide a simple Application Programming Interface (API) for average users to be able to build and train neural networks with ease, though it is not as complete and optimized as world-renowned libraries such as PyTorch and Tensorflow since it is mainly for learning purposes.

The building blocks of our project consist of arrays and many operations from Numpy, the most well-known Python library for scientific computing. Thanks to the library, we are able to achieve performance that is hundreds of times better than Python's built-in data types. The project also takes advantage of other libraries such as Scikit-learn or Keras for testing purposes.

In order to run the project, Python version 3.7 or newer is recommended, as it supports type annotations, f-string and various other techniques used in the code base. Libraries and dependencies can be installed using pip with the single command

```
pip3 install numpy scikit-learn tensorflow
```

Project layout and contents

The project's repository is hosted on Github and is accessible via its URL at https://github.com/hanzopgp/DeepLearningLib. It has the following layout:

```
./
 ├── nndiy/
 |    ├── __init__.py
 |    ├── activation.py
 |    ├── core.py
 |    ├── early_stopping.py
 |    ├── layer.py
 |    ├── loss.py
 |    ├── optimizer.py
 |    └── utils.py
 ├── experiences.py
 ├── unit_test.py
 └── README.md
```

in which:

- the folder `nndiy` contains the library's source code and is importable from another Python file using the familiar syntax `import nndiy` or `from nndiy import *`;
- `__init__.py` contains the class `Sequential` representing the neural network object itself;
- `core.py` contains essential abstract classes from which all other classes in the library inherit, including `Module` (one single layer of the network), `Activation` (activation layer), `Loss` (network's last layer), and `Optimizer`;
- `layer.py` contains classes that represent basic parameterized layers including `Linear`, `Convo1D` (1D convolutional layer), `MaxPool1D`, `Flatten` and `Dropout`;
- `activation.py` contains all classes representing activation layers inheriting from `Activation`, including `LeakyReLU`, `Identity` (where input is identical to output i.e. the linear activation layer), `ReLU`, `Sigmoid`, `Softmax`, and `Tanh`;
- `loss.py` contains implemented loss functions suitable for various learning tasks, including `(Sparse)BCE`, `(Sparse)CCE`, `SparseCCESoftmax`, `MAE`, `MSE`, and `RMSE`;
- `optimizer.py` contains some popular parameter optimizers including `GD`, `SGD`, `MGD`, and `Adam`;
- `early_stopping.py` contains a utility class implementing early-stopping functionality for early training termination;
- and `utils.py` contains some useful functions that are used across the library, notably `one_hot` which is essential to the sparse loss functions.

Besides, the code in the `experiences.py` file contains our experiments with our hand-crafted library using a different architecture and dataset each time, whose outputs are shown and

explained in the later parts of this report, and `unit_test.py` where we put our unit tests to support the project's development.

Unit testing

The development of the project was rather smooth since we understood most of the theories taught in class. The difficulty lies in the project's structure and optimization to be able to produce clean and maintainable code as well as to have a reasonable execution time. To achieve this goal, we first developed a set of unit tests at a less fine granularity than usual to quickly test the project's functionality as a whole rather than testing each individual function.

The tests are highly modulable and configurable, allowing us to test different network architectures on generated toy data and observe the output visually clearly. Below are some quick tests we did before introducing our library to more realistic problems.

```
===== SIMPLE CLASSIFICATION PROBLEM WITH 2 CLASSES =====
Testing 2 Gaussians, BCE loss, GD:
  Score: 1.0000 OK
Testing 2 Gaussians, BCE loss, SGD:
  Score: 1.0000 OK
Testing 2 Gaussians, BCE loss, MGD:
  Score: 1.0000 OK
Testing 2 Gaussians, Sparse BCE loss, GD:
  Score: 0.9700 OK
Testing 2 Gaussians, Sparse BCE loss, SGD:
  Score: 0.9750 OK
Testing 2 Gaussians, Sparse BCE loss, MGD:
  Score: 0.9500 OK
Testing 4 Gaussians, BCE loss, GD optim:
  Score: 1.0000 OK
Testing 4 Gaussians, BCE loss, SGD optim:
  Score: 1.0000 OK
Testing 4 Gaussians, BCE loss, MGD optim:
  Score: 1.0000 OK
Testing 4 Gaussians, Sparse BCE loss, GD optim:
  Score: 0.9000 OK
Testing 4 Gaussians, Sparse BCE loss, SGD optim:
  Score: 0.9300 OK
Testing 4 Gaussians, Sparse BCE loss, MGD optim:
  Score: 0.8700 OK
```

```
===== CLASSIFICATION WITH 4 CLASSES =====
Testing Vertical data, CCE, GD optim:
  Score: 0.9400 OK
Testing Vertical data, CCE, SGD optim:
  Score: 0.9250 OK
Testing Vertical data, CCE, MGD optim:
  Score: 0.9700 OK
Testing Vertical data, Sparse CCE, GD optim:
  Score: 0.9000 OK
Testing Vertical data, Sparse CCE, SGD optim:
  Score: 0.9450 OK
Testing Vertical data, Sparse CCE, MGD optim:
  Score: 0.9300 OK
```

```
===== REGRESSION PROBLEM =====
Testing Optimizer gd    Loss function mse:
  Score: 0.0164 OK
Testing Optimizer gd    Loss function mae:
  Score: 0.0949 OK
Testing Optimizer gd    Loss function rmse:
  Score: 0.0271 OK
Testing Optimizer sgd   Loss function mse:
  Score: 0.0095 OK
Testing Optimizer sgd   Loss function mae:
  Score: 0.0100 OK
Testing Optimizer sgd   Loss function rmse:
  Score: 0.0114 OK
Testing Optimizer mgd   Loss function mse:
  Score: 0.0086 OK
Testing Optimizer mgd   Loss function mae:
  Score: 0.0109 OK
Testing Optimizer mgd   Loss function rmse:
  Score: 0.0100 OK
```

Image classification using Multi-Layer Perceptron (MLP)

To first experiment with the library, we have decided to test its ability to classify hand-written digits from the MNIST dataset. The dataset contains 60,000 training and 10,000 testing grayscale images of dimension 28 by 28 pixels. Taking performance into account, we pick only 10,000 images for training and another 2,000 for validating. Our network has the following architecture:

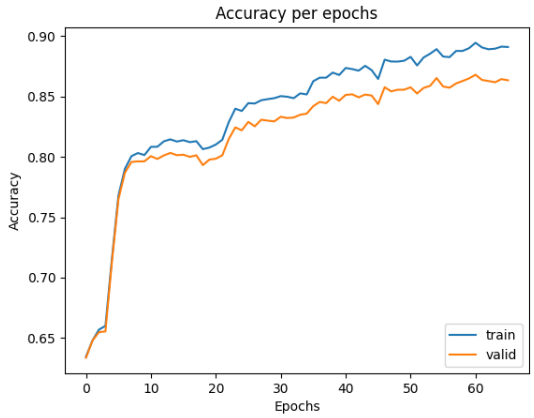
1. Linear layer with 784 dimensions (flattened image of 28 by 28) as input and 256 dimensions as output, hence 200,704 weights and 256 biases;
2. Tanh activation layer;
3. Linear layer going from 256 input to 128 input dimensions, hence 32,768 weights and 128 biases;
4. Tanh activation layer;
5. Linear layer of shape (128, 10) outputting 10 values associated to the 10 classes that this dataset has;
6. Softmax activation layer transforming the values obtained from the last linear layer to signify the prediction;
7. Sparse CCE Softmax loss computing the risk of incorrect predictions for this particular problem (multi-label, softmax output activation).

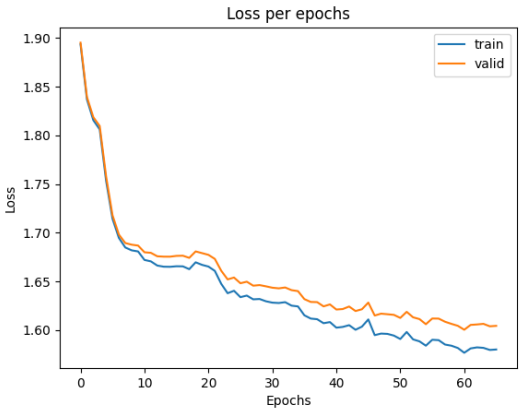
The network is compiled with the SGD optimizer and an initial learning rate of 0.001 and a very small rate of decay. From the training, we observe some interesting stats as shown below.

```
epoch : 025, train_acc : 0.825200, train_loss :  1.645687e+00, valid_acc : 0.808500, valid_loss :  1.660553e+00, learning_rate : 9.680492e-04
100%|| 10000/10000 [00:26<00:00, 383.95it/s]
epoch : 026, train_acc : 0.825000, train_loss :  1.645895e+00, valid_acc : 0.807700, valid_loss :  1.661449e+00, learning_rate : 9.655388e-04
100%|| 10000/10000 [00:24<00:00, 405.78it/s]
epoch : 027, train_acc : 0.825500, train_loss :  1.645142e+00, valid_acc : 0.806800, valid_loss :  1.661542e+00, learning_rate : 9.629388e-04
100%|| 10000/10000 [00:27<00:00, 363.37it/s]
epoch : 028, train_acc : 0.831500, train_loss :  1.637799e+00, valid_acc : 0.815000, valid_loss :  1.653075e+00, learning_rate : 9.602501e-04
100%|| 10000/10000 [00:25<00:00, 386.34it/s]
epoch : 029, train_acc : 0.840000, train_loss :  1.629902e+00, valid_acc : 0.822400, valid_loss :  1.644419e+00, learning_rate : 9.574734e-04
100%|| 10000/10000 [00:23<00:00, 417.22it/s]
epoch : 030, train_acc : 0.850600, train_loss :  1.620183e+00, valid_acc : 0.832400, valid_loss :  1.634375e+00, learning_rate : 9.546096e-04
100%|| 10000/10000 [00:25<00:00, 389.09it/s]
epoch : 031, train_acc : 0.857800, train_loss :  1.612837e+00, valid_acc : 0.841000, valid_loss :  1.626844e+00, learning_rate : 9.516595e-04
100%|| 10000/10000 [00:26<00:00, 380.90it/s]
epoch : 032, train_acc : 0.861900, train_loss :  1.608468e+00, valid_acc : 0.843100, valid_loss :  1.624770e+00, learning_rate : 9.486239e-04
100%|| 10000/10000 [00:25<00:00, 387.23it/s]
epoch : 033, train_acc : 0.869000, train_loss :  1.602088e+00, valid_acc : 0.848600, valid_loss :  1.618704e+00, learning_rate : 9.455037e-04
100%|| 10000/10000 [00:28<00:00, 352.01it/s]
epoch : 034, train_acc : 0.872500, train_loss :  1.598723e+00, valid_acc : 0.851000, valid_loss :  1.616157e+00, learning_rate : 9.422999e-04
100%|| 10000/10000 [00:29<00:00, 344.81it/s]
epoch : 035, train_acc : 0.876700, train_loss :  1.593568e+00, valid_acc : 0.854700, valid_loss :  1.611611e+00, learning_rate : 9.390134e-04
100%|| 10000/10000 [00:31<00:00, 313.53it/s]
epoch : 036, train_acc : 0.880400, train_loss :  1.589293e+00, valid_acc : 0.859400, valid_loss :  1.607732e+00, learning_rate : 9.356450e-04
100%|| 10000/10000 [00:30<00:00, 324.88it/s]
epoch : 037, train_acc : 0.882300, train_loss :  1.587010e+00, valid_acc : 0.859800, valid_loss :  1.606568e+00, learning_rate : 9.321959e-04
100%|| 10000/10000 [00:28<00:00, 347.63it/s]
epoch : 038, train_acc : 0.883400, train_loss :  1.586198e+00, valid_acc : 0.859700, valid_loss :  1.606073e+00, learning_rate : 9.286670e-04
100%|| 10000/10000 [00:28<00:00, 355.91it/s]
epoch : 039, train_acc : 0.887300, train_loss :  1.582093e+00, valid_acc : 0.863900, valid_loss :  1.603143e+00, learning_rate : 9.250592e-04
100%|| 10000/10000 [00:27<00:00, 370.26it/s]
epoch : 040, train_acc : 0.891300, train_loss :  1.576339e+00, valid_acc : 0.869800, valid_loss :  1.596474e+00, learning_rate : 9.213737e-04
100%|| 10000/10000 [00:25<00:00, 391.69it/s]
epoch : 041, train_acc : 0.888800, train_loss :  1.580517e+00, valid_acc : 0.864100, valid_loss :  1.601791e+00, learning_rate : 9.176115e-04
100%|| 10000/10000 [00:30<00:00, 331.21it/s]
epoch : 042, train_acc : 0.893000, train_loss :  1.574911e+00, valid_acc : 0.868400, valid_loss :  1.596793e+00, learning_rate : 9.137737e-04
100%|| 10000/10000 [00:27<00:00, 358.51it/s]
epoch : 043, train_acc : 0.892500, train_loss :  1.575893e+00, valid_acc : 0.867200, valid_loss :  1.598182e+00, learning_rate : 9.098613e-04
100%|| 10000/10000 [00:30<00:00, 333.04it/s]
epoch : 044, train_acc : 0.894700, train_loss :  1.573651e+00, valid_acc : 0.868800, valid_loss :  1.597095e+00, learning_rate : 9.058754e-04
100%|| 10000/10000 [00:24<00:00, 416.35it/s]
epoch : 045, train_acc : 0.896200, train_loss :  1.571439e+00, valid_acc : 0.870300, valid_loss :  1.595495e+00, learning_rate : 9.018173e-04
--> Early stopping triggered and best model returned from epoch number 40
```

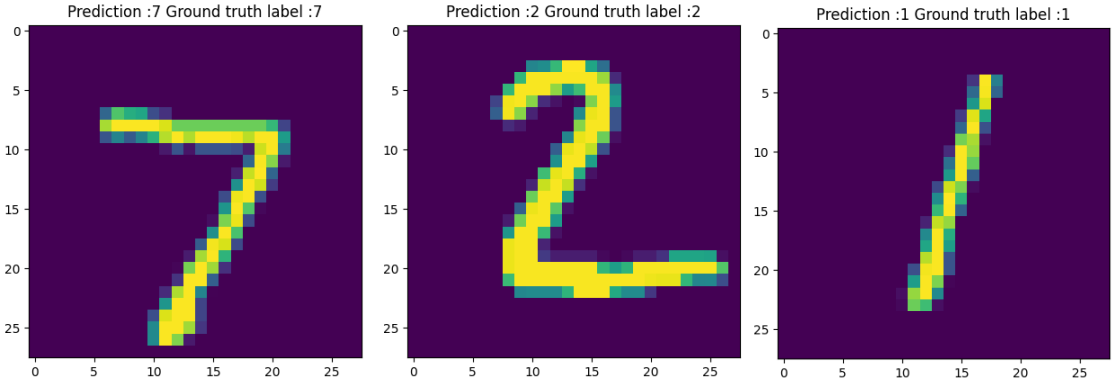
We can see that the loss and accuracy evolve quite well over time, with about 1-2% of gain in validation accuracy after each epoch. However, later epochs show that we have converged

to a potentially local minimum due to decaying learning rate, that even though the network still has room to improve itself until overfitting (89% of training accuracy), it has ceased to learn and early stopping was triggered at epoch number 45. The graphs below clearly show the network's improvement over time.



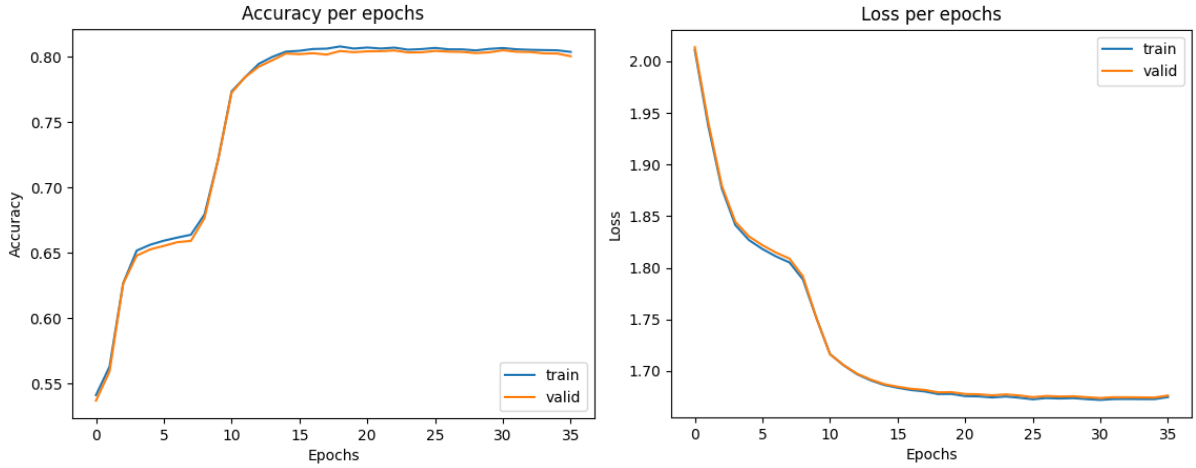


Here are some examples of predictions in the test set.

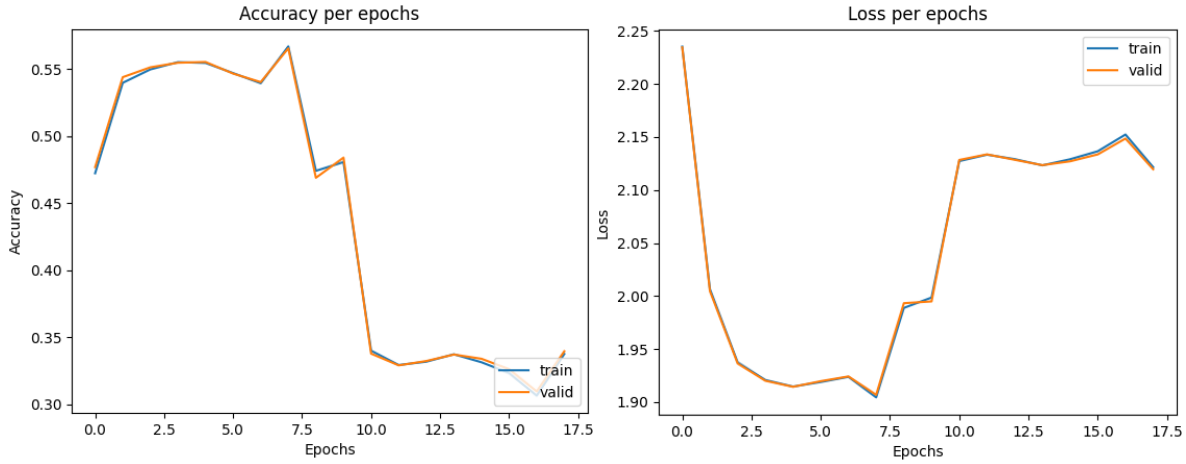


We decided to up the game by training the network with 48,000 data and 12,000 for validation using the same architecture. We observe this time that the training, despite much longer, provides definitively better and more stable results. With five times more data, accuracy and loss converge early, around the 15^{th} epoch at 80% of accurate predictions on the validation set. Although this ratio is 9% lower than that in the previous example, it should be noted that there are significantly more testing data this time, and the lower but still very good score also means that our network is more generalizable and far from overfitting if given more data to train on. The screenshots below show the evolution of training epochs as well as the its convergence on the plot.

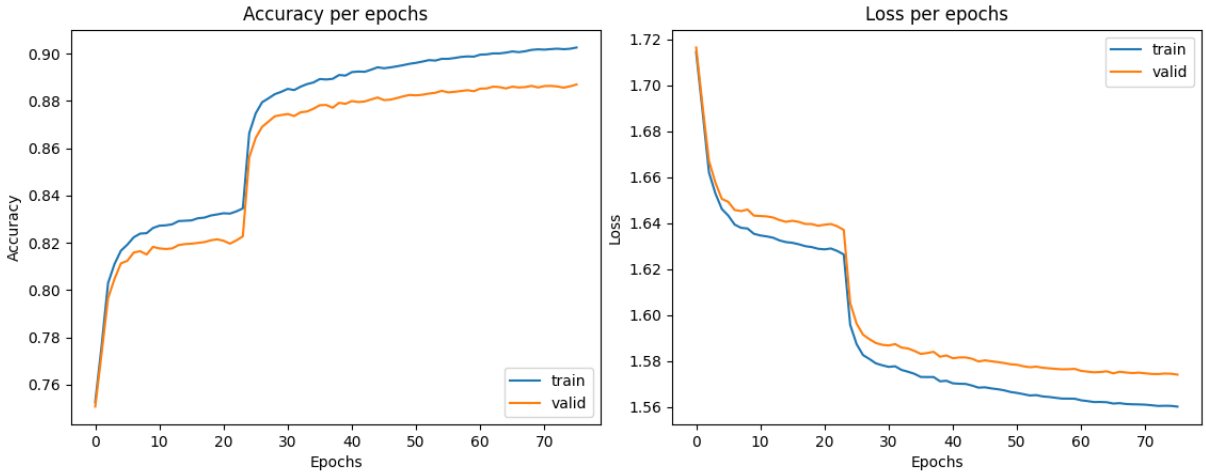
```
epoch : 026, train_acc : 0.806708, train_loss :  1.672613e+00, valid_acc : 0.804417, valid_loss :  1.674704e+00, learning_rate : 8.396837e-05
100%|| 48000/48000 [01:30<00:00, 532.91it/s]
epoch : 027, train_acc : 0.805625, train_loss :  1.673833e+00, valid_acc : 0.803833, valid_loss :  1.675803e+00, learning_rate : 8.284990e-05
100%|| 48000/48000 [01:24<00:00, 565.46it/s]
epoch : 028, train_acc : 0.805583, train_loss :  1.673397e+00, valid_acc : 0.803667, valid_loss :  1.675326e+00, learning_rate : 8.170602e-05
100%|| 48000/48000 [01:23<00:00, 575.82it/s]
epoch : 029, train_acc : 0.804792, train_loss :  1.673763e+00, valid_acc : 0.802750, valid_loss :  1.675544e+00, learning_rate : 8.053821e-05
100%|| 48000/48000 [1:07:10<00:00, 11.91it/s]
epoch : 030, train_acc : 0.805958, train_loss :  1.672775e+00, valid_acc : 0.803333, valid_loss :  1.674653e+00, learning_rate : 7.934799e-05
100%|| 48000/48000 [18:50<00:00, 42.47it/s]
epoch : 031, train_acc : 0.806625, train_loss :  1.672022e+00, valid_acc : 0.804917, valid_loss :  1.673860e+00, learning_rate : 7.813687e-05
100%|| 48000/48000 [01:47<00:00, 446.27it/s]
epoch : 032, train_acc : 0.805646, train_loss :  1.672840e+00, valid_acc : 0.803750, valid_loss :  1.674666e+00, learning_rate : 7.690637e-05
100%|| 48000/48000 [01:45<00:00, 453.15it/s]
epoch : 033, train_acc : 0.805250, train_loss :  1.672945e+00, valid_acc : 0.803583, valid_loss :  1.674580e+00, learning_rate : 7.565801e-05
100%|| 48000/48000 [03:47<00:00, 211.14it/s]
epoch : 034, train_acc : 0.805042, train_loss :  1.672819e+00, valid_acc : 0.802583, valid_loss :  1.674463e+00, learning_rate : 7.439332e-05
100%|| 48000/48000 [39:56<00:00, 20.03it/s]
epoch : 035, train_acc : 0.804875, train_loss :  1.672779e+00, valid_acc : 0.802417, valid_loss :  1.674358e+00, learning_rate : 7.311383e-05
100%|| 48000/48000 [01:51<00:00, 429.19it/s]
epoch : 036, train_acc : 0.803646, train_loss :  1.674708e+00, valid_acc : 0.800333, valid_loss :  1.676258e+00, learning_rate : 7.182105e-05
--> Early stopping triggered and best model returned from epoch number 26
```



Lastly for the MNIST classification problem with SGD optimization, we changed the activation function to ReLU to see if it can improve the performance. Interestingly, why being widely adapted to more learning problems than Tanh, ReLU has failed to learn the MNIST problem effectively. The graphs below show the fluctuated and unstable progress before early stopping was finally triggered.

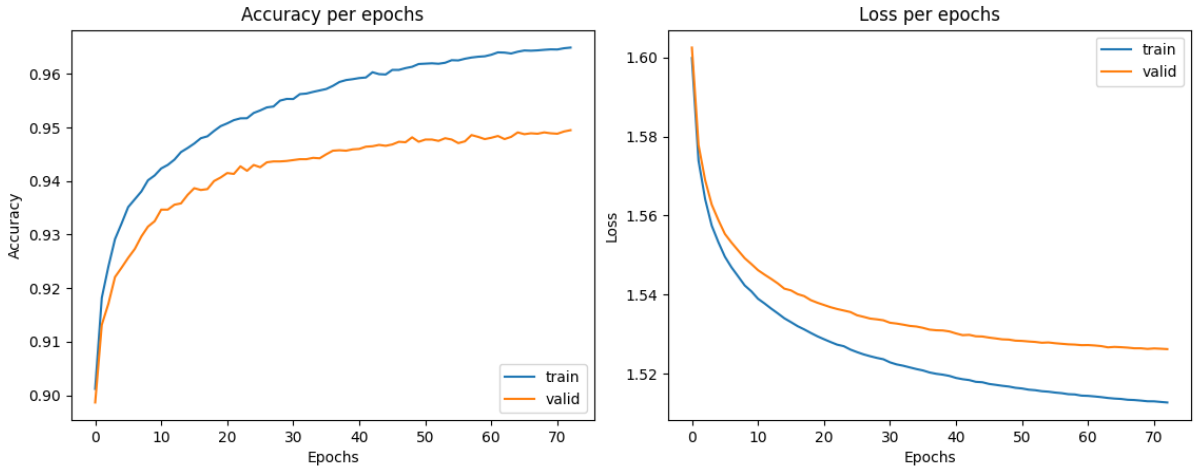


Having programmed one of the best optimizers out there, Adam, we must put it to the test to see it deserves the credit. Here are the graphs of Adam with ReLU on 48,000 training images.



As we can see, ReLU suddenly becomes feasible thanks to the help from Adam, with nearly 88% of validation accuracy and very stable convergence. We also observed an interesting sharp improvement at epoch 25. Our instinct tells that the event might have been caused by a sudden exit from a local minimum to arrive at a more optimal one or even better, the global minimum. Whatever the reason was, it is extremely difficult to find out.

As for the last test in this section, the results were unsurprisingly better with the Tanh activation function.



Since Adam is a stabilized variation of SGD, we observed a smooth convergence of the network to the overfitting mark of over 96% of training accuracy. This shows that even with so much data to train on, Adam was able to attain the limit of this particular dataset. To further extend the capability of this network, we could either introduce new training data (which we don't have), or add a couple of Dropout layers (which we have also implemented but not experimented).

Auto-encoder

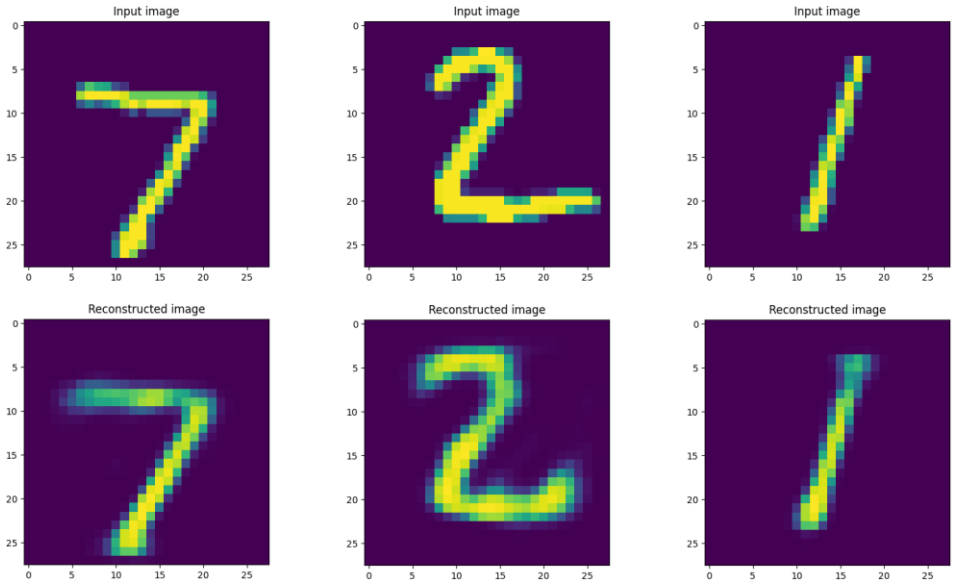
Always with the same MNIST dataset, we experimented the capability of our library with a slightly more difficult problem: compressing and decompressing images. For this purpose, we built an auto-encoder network architecture going from 784 dimensions to 256 in the first linear layer, 256 to 64 in the second, 64 to 256 in the third, and 256 back to 784 in the last layer, with the Tanh activation function for each hidden layer, and Sigmoid for the final layer which bends the output between 0 and 1 which is the same as the original data. With output inside this range, Binary Cross-Entropy is the most suitable loss function. The overall architecture is shown in the image below.

```
================================================================================
==> Network :
--> Layer 0 : Linear with parameters_shape = (784, 256) and activation : Tanh
--> Layer 1 : Linear with parameters_shape = (256, 64) and activation : Tanh
--> Layer 2 : Linear with parameters_shape = (64, 256) and activation : Tanh
--> Layer 3 : Linear with parameters_shape = (256, 784) and activation : Sigmoid
* Loss : BinaryCrossEntropy
* Optimizer : StochasticGradientDescent
* Total number of parameters : 435536
================================================================================
```

Training on 10,000 data was rapidly converged to very small loss value around zero, and early stopping was trigger at only epoch 16.

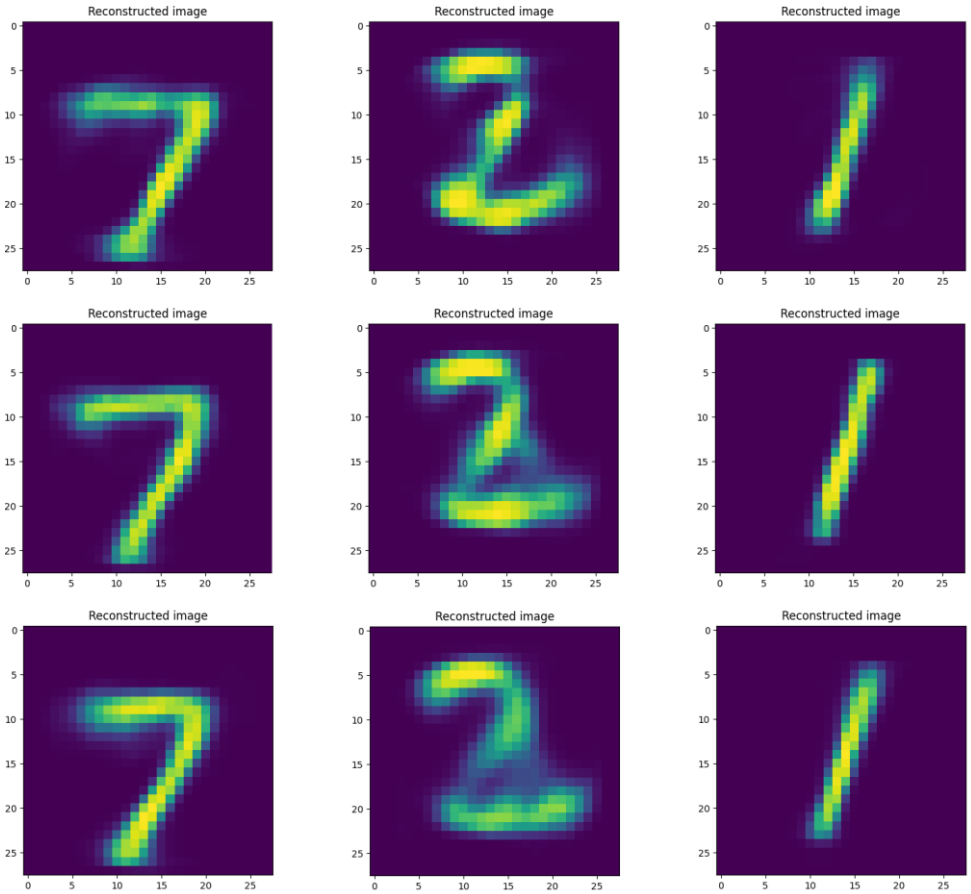
```
100%|| 10000/10000 [00:56<00:00, 176.15it/s]
epoch : 000, train_loss :  3.226485e-02, valid_loss :  3.198705e-02, learning_rate : 5.000000e-04
100%|| 10000/10000 [00:49<00:00, 203.47it/s]
epoch : 001, train_loss :  2.969540e-02, valid_loss :  2.970834e-02, learning_rate : 4.999500e-04
100%|| 10000/10000 [00:51<00:00, 193.88it/s]
epoch : 002, train_loss :  2.014304e-02, valid_loss :  2.025963e-02, learning_rate : 4.998500e-04
100%|| 10000/10000 [00:52<00:00, 188.97it/s]
epoch : 003, train_loss :  1.589428e-02, valid_loss :  1.576474e-02, learning_rate : 4.997001e-04
100%|| 10000/10000 [00:47<00:00, 209.51it/s]
epoch : 004, train_loss :  2.191490e-03, valid_loss :  2.202748e-03, learning_rate : 4.995003e-04
100%|| 10000/10000 [00:51<00:00, 192.70it/s]
epoch : 005, train_loss : -5.792479e-03, valid_loss : -5.787474e-03, learning_rate : 4.992507e-04
100%|| 10000/10000 [00:50<00:00, 199.81it/s]
epoch : 006, train_loss :  6.766600e-03, valid_loss :  6.541727e-03, learning_rate : 4.989513e-04
100%|| 10000/10000 [00:48<00:00, 204.41it/s]
epoch : 007, train_loss :  1.218925e-02, valid_loss :  1.234750e-02, learning_rate : 4.986023e-04
100%|| 10000/10000 [00:52<00:00, 191.06it/s]
epoch : 008, train_loss : -8.146290e-04, valid_loss : -8.396508e-04, learning_rate : 4.982037e-04
100%|| 10000/10000 [00:56<00:00, 177.27it/s]
epoch : 009, train_loss :  1.010395e-02, valid_loss :  1.040824e-02, learning_rate : 4.977558e-04
100%|| 10000/10000 [00:49<00:00, 201.31it/s]
epoch : 010, train_loss :  6.262956e-03, valid_loss :  6.479223e-03, learning_rate : 4.972585e-04
100%|| 10000/10000 [00:48<00:00, 207.58it/s]
epoch : 011, train_loss : -9.991373e-04, valid_loss : -7.778188e-04, learning_rate : 4.967121e-04
100%|| 10000/10000 [00:50<00:00, 197.92it/s]
epoch : 012, train_loss :  8.219304e-03, valid_loss :  8.471043e-03, learning_rate : 4.961168e-04
100%|| 10000/10000 [00:51<00:00, 195.25it/s]
epoch : 013, train_loss :  3.177561e-04, valid_loss :  4.154964e-04, learning_rate : 4.954727e-04
100%|| 10000/10000 [00:47<00:00, 209.02it/s]
epoch : 014, train_loss :  1.164666e-02, valid_loss :  1.189257e-02, learning_rate : 4.947800e-04
100%|| 10000/10000 [00:47<00:00, 209.37it/s]
epoch : 015, train_loss :  2.902550e-03, valid_loss :  3.133867e-03, learning_rate : 4.940389e-04
100%|| 10000/10000 [00:47<00:00, 209.14it/s]
epoch : 016, train_loss : -2.612117e-03, valid_loss : -2.391883e-03, learning_rate : 4.932497e-04
--> Early stopping triggered and best model returned from epoch number 11
```

As a result, reconstructing compressed images, although does not look identical to original ones, are clearly recognizable to human eyes, as shown in the examples below.



The ability to compress images with a very high ratio (784:64 or roughly 92%) without losing much information brings our implementation close to today's most popular image compression method, JPEG, where average ratio is about 10:1.

Since the results were satisfying, we continued to push the model further by trying out more aggressive compression ratios. The results below, with nearly identical configuration, show the reconstruction of the above examples using respectively 784:16 ratio trained on 10,000 data, 784:8 on 30,000, and 784:6 on 30,000.



As expected, noises as well as pixel saturation are much more evident as we heavily compress and reconstruct the images, especially with similar-looking classes such as 2 and 8, 3 and 8, 4 and 9, etc. However, most of the time our eyes are able to "de-noise" and clearly tell correctly the reconstructed digit from the 784:8 (99%) compression ratio, which is amazing! In a few paragraphs, we are going to put the auto-encoder to a more challenging test with some de-noising tasks.

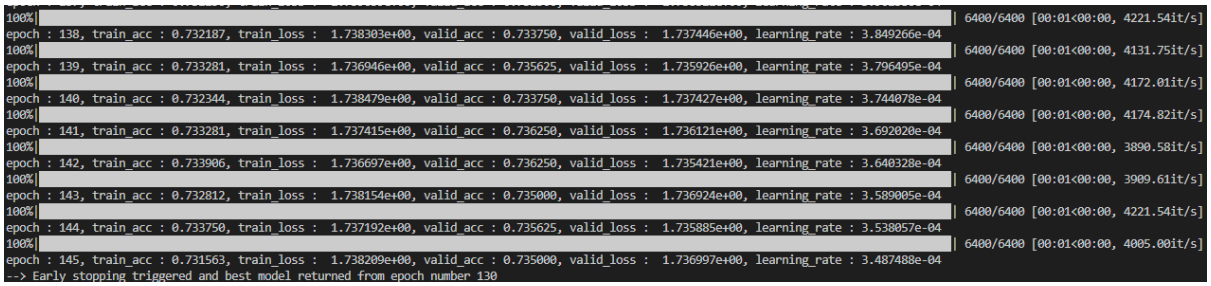
To better understand the magic behind data compression, we assumed the hypothesis that even at the highly compressed state (in the smallest hidden layer), each of the data is still very much representative for its own class and at the same time, data of the same class is tightly close together in the tiny latent space and distant from those of other classes, which is the reason why reconstructed images are visibly correct.

To test this theory, we built a MLP classifier similar to the last part of the report, changing the input size to match the shape of the compressed images. In particular, the architecture for classifying 64-pixel image includes:

1. Linear layer inputting 64 dimensions and outputting 32, Tanh activation;
2. Linear layer from 32 to 16 dimensions, Tanh activation;
3. Output linear layer from 16 to 10 dimensions, Softmax activation.

With SGD optimizer and Sparse CCE Softmax loss, for the first experiement, we achieved a decent 78% validation accuracy after 200 epochs of training on 6,400 compressed inputs. Toward the end the network has possibly been sucked into a local minimum since the network stopped improving at epoch 150.

Since 784:8 compression trained on 30,000 data showed good potential as seen in the last part, another 6,400 images were compressed using this model and used as training data for the MLP classifier. Without much surprise, its performance was very good.



While the final compressed data has reduced 8 times in size, the classifier only lost a few percent compared to the last one. If given more data and better hyperparameter tuning, we are certain that this model would outperform the last one.

Reducing the latent space even more, down to 6 dimensions, had counter-benefited the performance, as validation accuracy was dropped down to 46% before early stopping was triggered. We therefore concluded that the 784:8 compression ratio was optimal to the MNIST dataset.

The last and most difficult task that we used auto-encoder for is image de-noising. For this task, we proceed with the following steps:

1. Determine the amount of noise;
2. Apply noise to an image by sampling that amount of pixel randomly and change the selected pixels' value randomly;
3. Build and train the network using noisy images as input and original images as output.

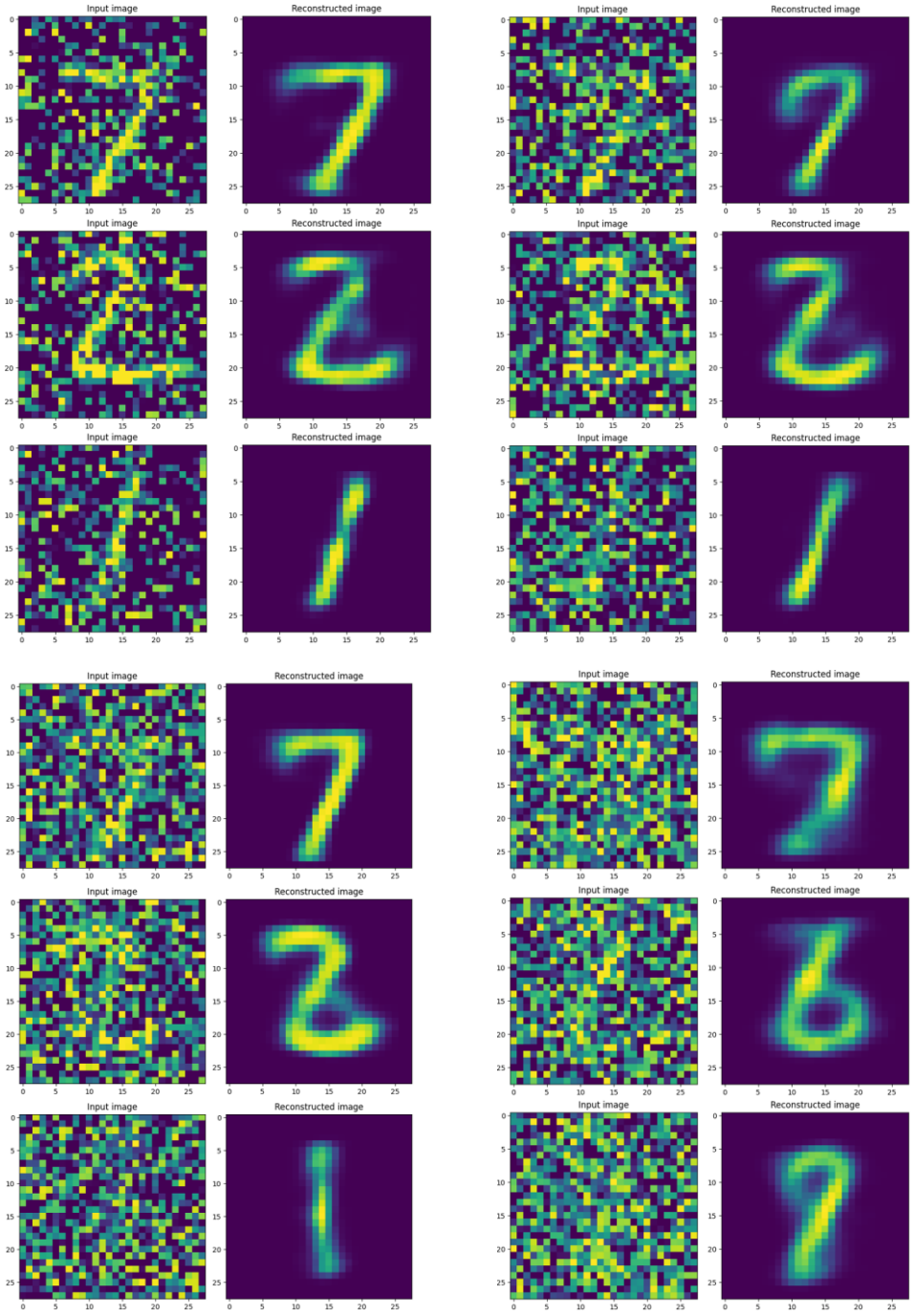
The network architecture varies in terms of amount of noise applied to input images. For instance, with inputs having only 10% of noises, the network has only 2 hidden layers:

```
==================================================================================================
==> Network :
--> Layer 0 : Linear with parameters_shape = (784, 256) and activation : Tanh
--> Layer 1 : Linear with parameters_shape = (256, 128) and activation : Tanh
--> Layer 2 : Linear with parameters_shape = (128, 256) and activation : Tanh
--> Layer 3 : Linear with parameters_shape = (256, 784) and activation : Sigmoid
* Loss : BinaryCrossEntropy
* Optimizer : StochasticGradientDescent
* Total number of parameters : 468368
==================================================================================================
```

On the other hand, we doubled the number of hidden layers and switched their activation function for complex inputs having more than 80% of noises:

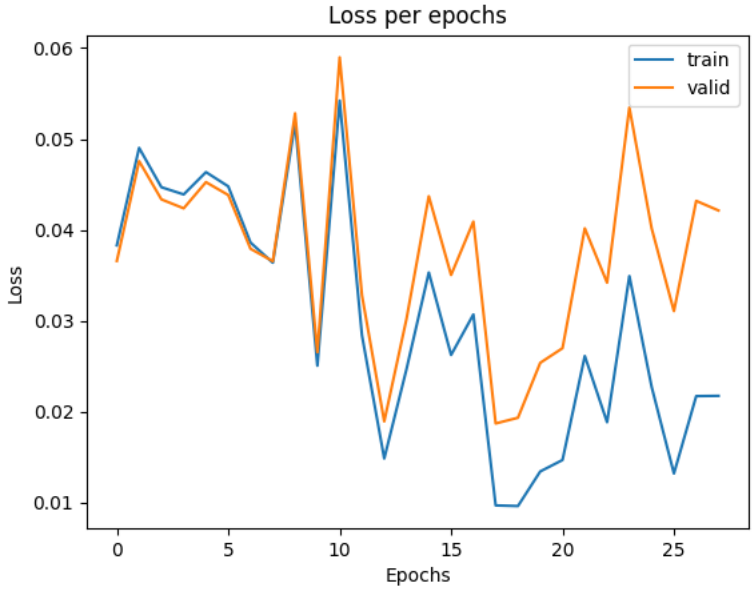
```
================================================================================================
==> Network :
--> Layer 0 : Linear with parameters_shape = (784, 256) and activation : ReLU
--> Layer 1 : Linear with parameters_shape = (256, 180) and activation : ReLU
--> Layer 2 : Linear with parameters_shape = (180, 128) and activation : ReLU
--> Layer 3 : Linear with parameters_shape = (128, 180) and activation : ReLU
--> Layer 4 : Linear with parameters_shape = (180, 256) and activation : ReLU
--> Layer 5 : Linear with parameters_shape = (256, 784) and activation : Sigmoid
* Loss : BinaryCrossEntropy
* Optimizer : StochasticGradientDescent
* Total number of parameters : 541432
================================================================================================
```

The image below captures the inputs and outputs of networks trained with respectively 40%, 60%. 70%, and 80% of noise.

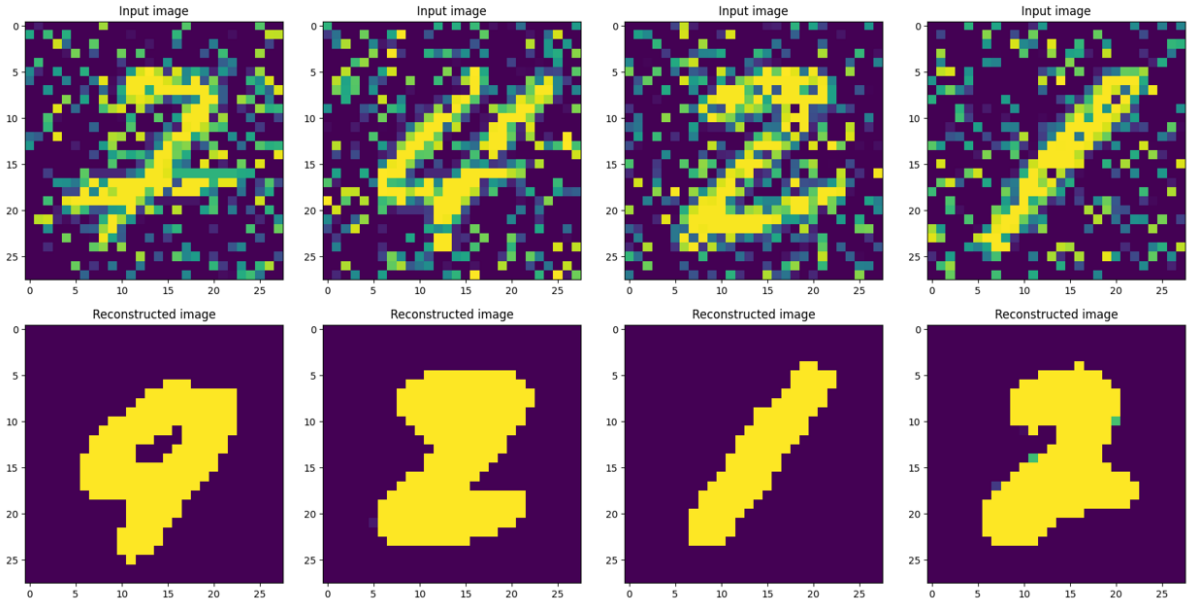


It is clear that the more noise applied to an input, the less clear is the output, as resulted in the second and third images with 80% of noise applied. However, even with that much noise, out network was able to trigger early stopping after only 27 epochs when validation BCE loss was very low. The result was a model that can process a very noisy image which human eyes can hardly see.

```
100%|| 48000/48000 [04:07<00:00, 194.26it/s]
epoch : 021, train_loss :  2.613991e-02, valid_loss :  4.017356e-02, learning_rate : 4.456437e-05
100%|| 48000/48000 [04:01<00:00, 198.95it/s]
epoch : 022, train_loss :  1.885849e-02, valid_loss :  3.419717e-02, learning_rate : 4.407950e-05
100%|| 48000/48000 [04:02<00:00, 198.04it/s]
epoch : 023, train_loss :  3.493210e-02, valid_loss :  5.345223e-02, learning_rate : 4.357835e-05
100%|| 48000/48000 [05:20<00:00, 149.74it/s]
epoch : 024, train_loss :  2.274197e-02, valid_loss :  4.017110e-02, learning_rate : 4.306161e-05
100%|| 48000/48000 [06:03<00:00, 132.00it/s]
epoch : 025, train_loss :  1.321912e-02, valid_loss :  3.107043e-02, learning_rate : 4.252998e-05
100%|| 48000/48000 [05:44<00:00, 139.32it/s]
epoch : 026, train_loss :  2.172948e-02, valid_loss :  4.319406e-02, learning_rate : 4.198419e-05
100%|| 48000/48000 [05:38<00:00, 141.63it/s]
epoch : 027, train_loss :  2.175450e-02, valid_loss :  4.213668e-02, learning_rate : 4.142495e-05
--> Early stopping triggered and best model returned from epoch number 12
```



The only problem with this approach was the non-generalizability of the model. We tried applying the 70% model to inputs having less amount of noise (i.e., a less difficult problem) but it failed to produce a satisfying output as seen below. It is possible that the model has also learned the fixed amount of noise therefore underperformed with easier inputs than those during training.



If we had trained a more complex network (more neurons and hidden layers) using randomly noised images, perhaps we would obtain a much better and generalized model.

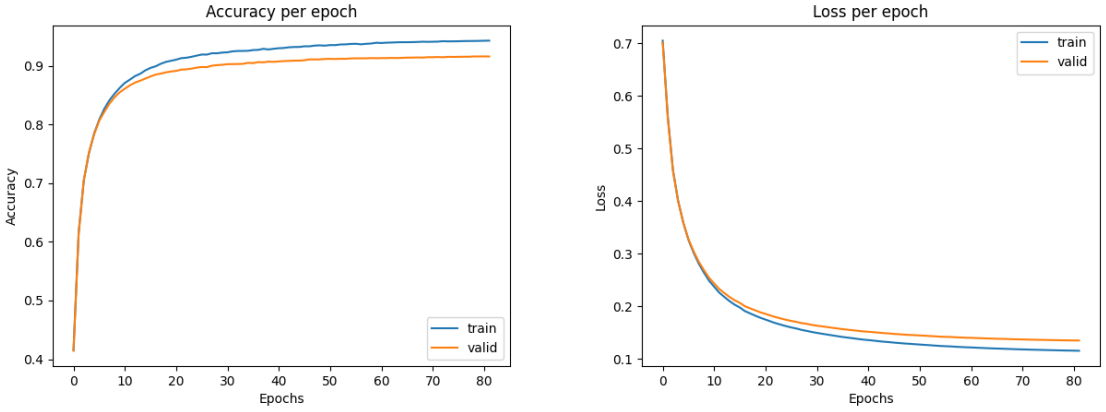
Convolution Neural Network

We felt that the library would not be complete without at least a convolutional layer class which would help learning images and signals a lot better with greater accuracy. Therefore, we have also implemented the `Convo1D`, `MaxPool1D`, `Flatten`, and `Dropout` classes as building blocks for convolution neural networks (CNN).

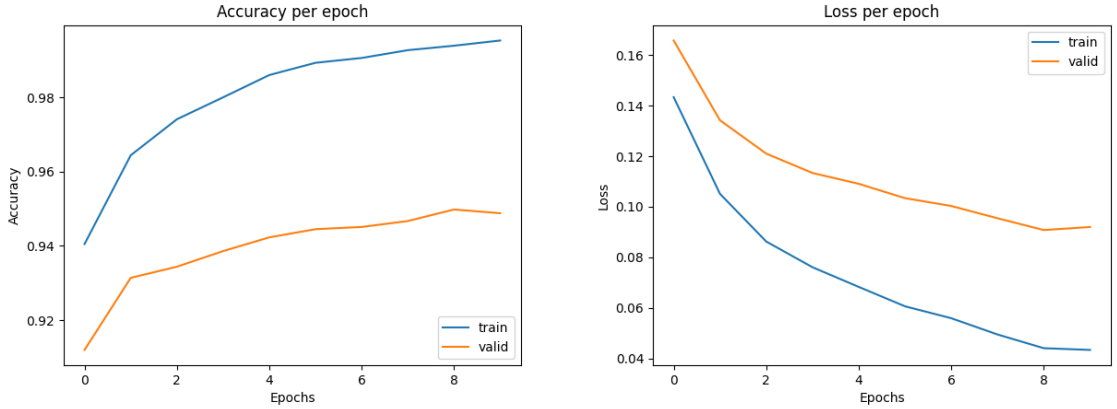
To experiment with our homemade CNN, we tested with the MNIST dataset again, this time flattening the images before passing them to the network as 1D signals, and compare the findings between SGD and Adam optimizers, since they are both popular in many CNN implementations found online. The network architecture is similar across all tests:

1. 1D convolutional layer with input size of 256 pixels * 1 channel, kernel size 3, 32 kernels, and 1-step stride;
2. Max pooling layer with kernel size 2, and 2-step stride;
3. Flatten layer;
4. Linear layer taking inputs of 4064 dimensions outputting 100 dimensions;
5. An activation function;
6. Linear layer taking inputs of 100 dimensions outputting 10 dimensions;
7. Categorical Cross Entropy loss.

We started off with SGD optimizer and ReLU for the fifth layer, which had been the least stable combination in our first MLP classification test. This time using CNN, we achieved convergence, unsurprisingly, at around 91% validation accuracy. By switching from ReLU to Tanh, the model achieved identical results without any improvement. We then safely concluded that the powerful capability of CNN over this rather simple task successfully minimized the difference between these two activation functions.



However, by turning from SGD to Adam, we saw a tremendous improvement in the learning, as the model converged rapidly to the overfitting point. Training accuracy reached 99% after a few epochs while validation accuracy stayed at 94-95%. The MNIST was quickly solved and there was nothing left for the model to learn.



One caveat during the development of CNN was the number of for loops required to make the algorithm worked, since the size of inputs and outputs were most of the time unbalanced and therefore impossible to be calculated directly with Numpy.

With pure Python, for loops are very slow which rendered the experiments impossible to finish even a few epochs. Therefore, we implemented the commonly used solution using Numba in all of the classes concerning CNN. By moving some implementation out of the class methods to static methods and adding Numba's decorator, execution speed is shortened by at least 50 times.

Conclusion

Although this mini-implementation of neural network is nowhere as robust and optimized as some of the most popular libraries such as Tensorflow or PyTorch, the process of developing it has helped us a lot to better understand the complexity of the algorithms needed for machines to learn. Most of the time, even if our library is much more complete than now, Tensorflow and PyTorch would still be the default choice for building a neural network. However, since these two are sometimes too heavily implemented, the burden they put on less capable machines such as embedded systems or mobile devices can make them the second choice to our homemade library.

Even after the project has been graded, we are going to keep developing the library and going back to it as a great pedagogical resource for further Machine Learning researches. As an endnote for the report, a few possible improvements/optimizations can be made in the future include:

- 2D convolutional with max and average pooling;
- Migrating methods with for loops to Numba (SGD and MGD for example);
- Multi-threading;
- Pushing calculations to the GPU;
- Forgetting about Numba, using C++ instead;
- etc...